第16回

形式的検証ツールの効果的な利用法

――システム・レベル記述どうしの等価性を検証する

藤田昌宏

ここでは，商用の等価性検証ツールを用いて，システム・レベル記述（SystemC）どうしの等価性を判定する．実際の等価性検証では，初期化や記述の読み込みなどの前処理の時間が必要になる．また，ワード・レベル検証を設計全体に適用できるかどうかによっても処理時間が大幅に変わる．（編集部）

今回と次回は，C言語やRTLのハードウェア記述言語で表現された設計記述どうしの等価性を，商用の等価性検証ツールを使って判定してみます．対象としては，アルゴリズムの表現として記述されたSystemC記述と，同記述に対する計算式の簡単化，並列化，およびパイプライン化を人手で行った設計の間の等価性を検証していきます．

● 順序回路検証におけるレイテンシとスループットを指定

前回も説明したように，順序回路では，フリップフロップなどで実現された内部状態が存在します．出力値は，入力値とその内部状態の値で決定されます．入力が入ってから内部状態に依存した計算を行い，一定時刻後に出力が出てくることになります．また，ハードウェアは，「入力から出力を計算するという処理を無限に繰り返す」というように動作するのが一般的です（リアクティブ・システム）．したがって，二つの順序回路が等価であるか否かを調べるためには，「どのタイミングの入力に対する出力がどのタイミングで出てくるか」という情報を設計者がツールに与えてやる必要があります．通常，等価性検証ツールでは，これらの情報はレイテンシとスループットとして与えます．

前回も示しましたが，レイテンシとスループットは**図1**のようなタイミングの指定に対応しています．スループットは，ある入力が入ってから次の入力が入るまでの間隔をクロック数で指定します．レイテンシは，一つの入力が入り，その入力に対応する出力が出てくるまでの時間であり，これもクロック数で指定します．一つの入力が入ってから，レイテンシ分だけ経過した時刻で出力が等価か否かを調べることになります．

等価性検証では，二つの設計記述の等価性を検証します．一般に，片方は正しい（あるいは，正しいと考えている）設計であり，これは「仕様」となります．これに対して，もう一方の設計が仕様を実現する「実装」であると考えられます．つまり，仕様と等価であることを示すことで，実装の正しさを証明するのです．上のレイテンシとスループットは仕様と実装のそれぞれに対して指定する必要があります．

仕様として，アルゴリズムを表現するために通常のソフ

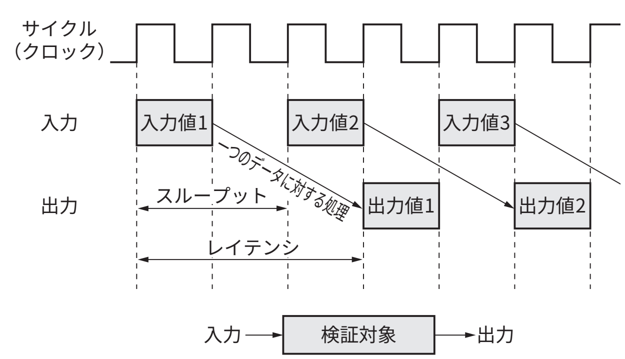


図1　レイテンシとスループット
一つのデータに対する計算時間が「レイテンシ」，入力データ間隔が「スループット」である

KeyWord　等価性検証，システム・レベル記述，SystemC，レイテンシ，スループット，IDCT，SLEC，ビヘイビア合成，パイプライン，リタイミング

トウェアと同じように記述されたものを利用する場合，すべての計算は瞬時に終了する形となり，1時刻で計算が終了し，かつ各入力は順に処理されていくことになります．つまり，レイテンシもスループットも1となります．しかし，より複雑な動作を表現し，部分的な動作の間の時間順序関係があるような記述を仕様とする場合には，実装だけでなく仕様についても，**図1**に示すレイテンシとスループットを指定する必要があります．

SystemCを利用するビヘイビア記述や，より上位のシステム・レベル設計記述では，かならずしもクロックという概念はありません．そのような場合，時刻を進める文として`wait()`を利用します．`wait()`文があると，時刻が一つ進むことを意味し，その`wait()`文の前の記述と後の記述では別の時刻の動作を表していることになります．したがって，**図2**に示すようにレイテンシが3であるSystemC記述には，実行のどのパスをたどっても，`wait()`文が2回実行されるようになっているはずです．

等価性検証では，仕様はこれまで正しいと考えられている設計なら何でもよいわけですから，一度等価であると判定された実装は，次の等価性検証では仕様として利用できます．つまり，設計記述を次々に改良し，詳細化していくような設計手法では，**図3**に示すように設計の各ステップの間で次々に等価性を検証していくことで，最終的な設計といちばん最初の仕様が等価であることを示せます．**図3**の上側に示す一連の等価性検証がすべて等価であると判定されれば，最初の仕様記述と最後の設計（実装）記述Nは等価であると言えます．

もちろん，**図3**の下側に示す最初と最後の記述の間の等価性検証を行ってもかまいません．しかし一般に等価性検証では，比較している二つの設計記述の差が小さいほど効率良く検証できます．例えば二つの記述はほぼ同じで，一つの記述ブロックに対してのみ，数式の最適化処理などが施されているだけの差の場合，差のある記述ブロックだけをチェックすればよいので，高速に検証できます．一般に記述の差が少ない場合には，多くの内部のフリップフロップが二つの設計データの間で同じ（あるいは対応する）状態を表していることが多く，その対応がとれる部分については，順序回路ではなく組み合わせ回路の検証手法を適用できます．そのため，等価性検証を大幅に効率化できます．

以下では，米国Calypto Design Systems社の「SLEC」と呼ばれる等価性検証ツールを利用しながら，SystemCによるシステム・レベル設計記述どうしの等価性検証，ならびにリタイミングなどのRTLの設計最適化に関係した等価性検証について見ていきたいと思います．

● システム・レベル記述どうしの等価性を検証

ここでは簡単でわかりやすい例として，画像処理などでよく使われるIDCT（逆離散コサイン変換）を考えます．典型的なソフトウェアとしてのプログラムをほぼそのままSystemC記述で表現したものを**リスト1**に示します（この名まえをIDCT0とする）．これを仕様（出発点）として，個々の算術式レベルの最適化や並列動作の導入，さらにはパイプライン化などを行った記述との等価性検証を実際に行いたいと思います．**リスト1**の記述では，関数`idctrow`と`idctcol`が各8回ずつ，異なる引き数値で順に実行されています．関数`idctrow`と`idctcol`はそれぞれ数十行の算術式から構成されています．SystemC記述にするために，`SC_METHOD`などの`SC`で始まるSystemC用の関数で全体が表現されています．この記述では`wait()`文はなく，すべての実行が瞬時に終了するという形になっています．このように1回だけの処理をすべて瞬時に行うような場合，

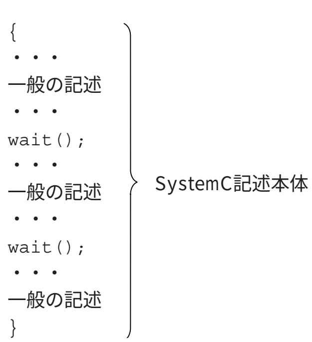


図2　SystemCの場合，時刻が進むことはwait()文で指定する
レイテンシが3の記述．どのパスをたどっても，`wait()`文が2回実行されるようになっている

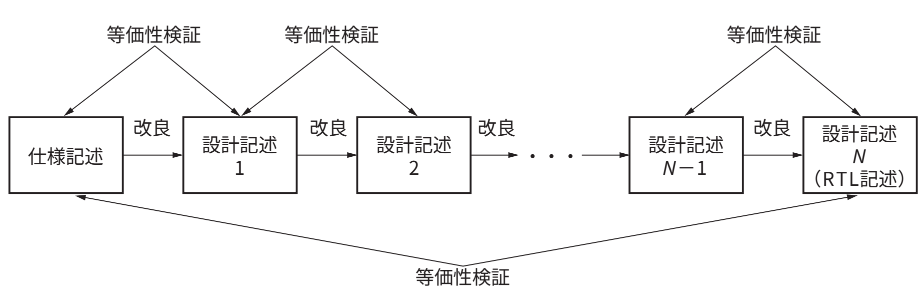


図3　設計の繰り返し改良・詳細化と等価性検証
実際の設計では，設計記述を次々に改良し，詳細化していくことが多い

Calypto社のツールでは，レイテンシやスループットをとくに指定する必要がありません．

まず，**リスト1**の記述どうしの等価性，つまり同じ記述の間の等価性を検証してみます．等価なのは当然なのですが，ツールがどのように処理するかを説明するうえでわかりやすいので，実行してみます．等価性を検証する手順ですが，以下の処理を行ったあと，検証することになります．

- 仕様（ここではSpecと呼ぶ）の読み込み
- 実装（ここではImplと呼ぶ）の読み込み
- 必要に応じてレイテンシやスループットの指定
- 必要に応じて仕様や実装のリセット条件の指定
- そのほか，もしあれば検証に有効な情報の入力（内部フリップフロップの対応など）

これらはすべてコマンドの形になっているので，コマンドを組み合わせたスクリプトを作成し，それをツールに与えることで等価性が検証されます．リセット条件は，仕様

リスト1　IDCTの記述（IDCT0）

```

#include "systemc.h"

SC_MODULE(spec0){

<定数定義部は省略>

  sc_in<short> blk_i[64];
  sc_out<short> blk_o[64];
  short blk[64];

  SC_CTOR(spec0){
    SC_METHOD(update);
  }

  void idctrow(int i){
    int x0, x1, x2, x3, x4, x5, x6, x7, x8;

    x1 = blk_i[i*8+4]<<11;
    x2 = blk_i[i*8+6];
    x3 = blk_i[i*8+2];
    x4 = blk_i[i*8+1];
    x5 = blk_i[i*8+7];
    x6 = blk_i[i*8+5];
    x7 = blk_i[i*8+3];
    x0 = (blk_i[i*8]<<11) + 128;

    x8 = W7*(x4+x5);
    x4 = x8 + (W1-W7)*x4;
    x5 = x8 - (W1+W7)*x5;
    x8 = W3*(x6+x7);
    x6 = x8 - (W3-W5)*x6;
    x7 = x8 - (W3+W5)*x7;

    x8 = x0 + x1;
    x0 -= x1;
    x1 = W6*(x3+x2);
    x2 = x1 - (W2+W6)*x2;
    x3 = x1 + (W2-W6)*x3;
    x1 = x4 + x6;
    x4 -= x6;
    x6 = x5 + x7;
    x5 -= x7;

    x7 = x8 + x3;
    x8 -= x3;
    x3 = x0 + x2;
    x0 -= x2;
    x2 = (181*(x4+x5)+128)>>8;
    x4 = (181*(x4-x5)+128)>>8;

    blk[i*8] = (x7+x1)>>8;
    blk[i*8+1] = (x3+x2)>>8;
    blk[i*8+2] = (x0+x4)>>8;
    blk[i*8+3] = (x8+x6)>>8;
    blk[i*8+4] = (x8-x6)>>8;
    blk[i*8+5] = (x0-x4)>>8;
    blk[i*8+6] = (x3-x2)>>8;
    blk[i*8+7] = (x7-x1)>>8;
  }

  void idctcol(int i){
    int x0, x1, x2, x3, x4, x5, x6, x7, x8;
    int y0, y1, y2, y3, y4, y5, y6, y7;

    x1 = blk[8*4+i]<<8;
    x2 = blk[8*6+i];
    x3 = blk[8*2+i];
    x4 = blk[8*1+i];
    x5 = blk[8*7+i];
    x6 = blk[8*5+i];
    x7 = blk[8*3+i];
    x0 = (blk[8*0+i]<<8) + 8192;

    x8 = W7*(x4+x5) + 4;
    x4 = (x8+(W1-W7)*x4)>>3;
    x5 = (x8-(W1+W7)*x5)>>3;
    x8 = W3*(x6+x7) + 4;
    x6 = (x8-(W3-W5)*x6)>>3;
    x7 = (x8-(W3+W5)*x7)>>3;

    x8 = x0 + x1;
    x0 -= x1;
    x1 = W6*(x3+x2) + 4;
    x2 = (x1-(W2+W6)*x2)>>3;
    x3 = (x1+(W2-W6)*x3)>>3;
    x1 = x4 + x6;
    x4 -= x6;
    x6 = x5 + x7;
    x5 -= x7;

    x7 = x8 + x3;
    x8 -= x3;
    x3 = x0 + x2;
    x0 -= x2;
    x2 = (181*(x4+x5)+128)>>8;
    x4 = (181*(x4-x5)+128)>>8;

    y0 = (x7+x1)>>14;
    y1 = (x3+x2)>>14;
    y2 = (x0+x4)>>14;
    y3 = (x8+x6)>>14;
    y4 = (x8-x6)>>14;
    y5 = (x0-x4)>>14;
    y6 = (x3-x2)>>14;
    y7 = (x7-x1)>>14;

    blk_o[8*0+i] = (y0<-256) ? -256 : ((y0>255) ? 255 : y0);
    blk_o[8*1+i] = (y1<-256) ? -256 : ((y1>255) ? 255 : y1);
    blk_o[8*2+i] = (y2<-256) ? -256 : ((y2>255) ? 255 : y2);
    blk_o[8*3+i] = (y3<-256) ? -256 : ((y3>255) ? 255 : y3);
    blk_o[8*4+i] = (y4<-256) ? -256 : ((y4>255) ? 255 : y4);
    blk_o[8*5+i] = (y5<-256) ? -256 : ((y5>255) ? 255 : y5);
    blk_o[8*6+i] = (y6<-256) ? -256 : ((y6>255) ? 255 : y6);
    blk_o[8*7+i] = (y7<-256) ? -256 : ((y7>255) ? 255 : y7);
  }

  void update(){
    for(int i=0; i<8; i++)
      idctrow(i);
    for(int i=0; i<8; i++)
      idctcol(i);
  }
};
```

や実装の記述にリセット信号がある場合にそれらをどのように制御するとリセットとなるか，また，通常の実行時にはどのように制御するべきかなどを指定します．このツールでは，指定されたリセット処理を仕様と実装の両方について実行した後の動作について，等価性の検証を行います．

リスト1の記述どうしの等価性検証では，レイテンシやスループットを指定する必要はなく，仕様と実装の記述が格納されているファイル名を指定して読み込み，検証を開始します．**リスト1**ではリセット条件もないため，リセットに関する指定は必要ありません．なお，以下の等価性検証では，3.2GHz動作のXeonプロセッサを搭載したパソコンの上で本ツールを稼働させています．

● 等価性検証にかかった時間の最初の15秒は前処理

リスト1の記述どうしの等価性検証を本ツールで行うと，約18秒で「等価である」という結果が得られます．等価なのは当然なのですが，その処理に18秒かかっていることになります．本ツールでは実行トレースなどの情報が表示されるので，それを少し見ていきましょう．

本ツールが起動されると，まず初期化を行ったあと，仕様と実装の記述を読み込みます．それに約8秒かかっています．次に，不等価であることが自明なケースを除去するために，内部で簡単なシミュレーションを行っています．もし，このシミュレーションで不等価であるとわかれば，そこで処理は終了します．このシミュレーションは1秒以内に終了しています．シミュレーションでは「等価であった」（実際には「不等価であるとは言えなかった」）ということなので，ここからほんとうの意味で等価性検証が始まります．そのため，読み込んだ仕様と実装の記述をツールが処理しやすいように少し変換します．これに数秒かかり，ツールの起動からここまでで合計約15秒かかっています．

つまり，どのような記述どうしの等価性検証でも，**リスト1**に示すくらいの量（百数十行の記述）を処理するのに，それくらいの時間がかかるということです．ですから，これを除いた処理時間がほんとうの意味での等価性検証に要している時間であると言うこともできます．今回の処理には18秒かかっているので，実際の等価性検証にかかるのは3秒くらいであると言えます．

順序回路どうしの比較であっても，フリップフロップや状態変数どうしの対応が完全にとれる場合は，組み合わせ回路の比較問題に帰着させることができます．また，対応が完全にはとれなくても，部分的にとれるのであれば，それを利用して等価性検証を効率化できます．検証時間を削減する場合，この対応づけは非常に重要な要素です．そこで，まずフリップフロップなどの内部記憶素子（変数）の対応がとれないかどうかを調べていきます．**リスト1**の記述どうしを比較する場合には，この処理によってすべての対応がとれることがわかるので，実際の検証は組み合わせ回路どうしの比較として実行されます．**リスト1**の記述の場合，対応をとるのに2秒ほどかかっているようです．

このように，順序回路に対応した等価性検証ツールには，効率的・効果的な組み合わせ回路の等価性検証の機能が組み込まれています．その意味では，順序回路に対する等価性検証ツールを，組み合わせ回路に対する等価性検証ツールとして利用することも可能です．以上の説明からわかるように，**リスト1**くらいの記述の等価性検証では，前処理に15秒くらいかかり，またフリップフロップの対応がとれるかどうかのチェックに2秒程度かかるということになります．このことを頭に入れておいてください．

● 算術式の変換操作はビヘイビア合成に相当

リスト1に改良を施したようすを**図4**に示します．

まず，IDCT1はIDCT0の中の関数`idctorow`と`idctcol`の中の算術式レベルの最適化や変換を行ったもので，総称してIDCT1と呼ぶことにします．この段階ではまだ，`wait()`文は挿入されていません．IDCT0とIDCT1の間の等価性検証は，上記とまったく同じ操作で行えます．変換のしかたによって，検証時間は20秒程度から50秒程度まで変化します．なお，前処理はどの検証でも必要となるので，この検証時間には上述の前処理時間である15秒が含まれていることになります．

また，この算術式の変換操作は，ハードウェアのビヘイビア記述（動作記述）をRTL記述に詳細化する処理と見なすこともできます．これは，一般にハイレベル合成（高位合成），あるいはビヘイビア合成（動作合成）と呼ばれる処理です．各実行文（算術式）をどのタイミング（クロック）で実行するか，すなわち「スケジューリング」と，各算術演算をどの演算器で実行するかを決める作業，すなわち「アロケーション・バインディング」が主となります．

スケジューリングは，記述に`wait()`文を挿入することで指定できます．アロケーション・バインディングについては，各演算器ごとに関数を用意し，同じ演算器を利用す

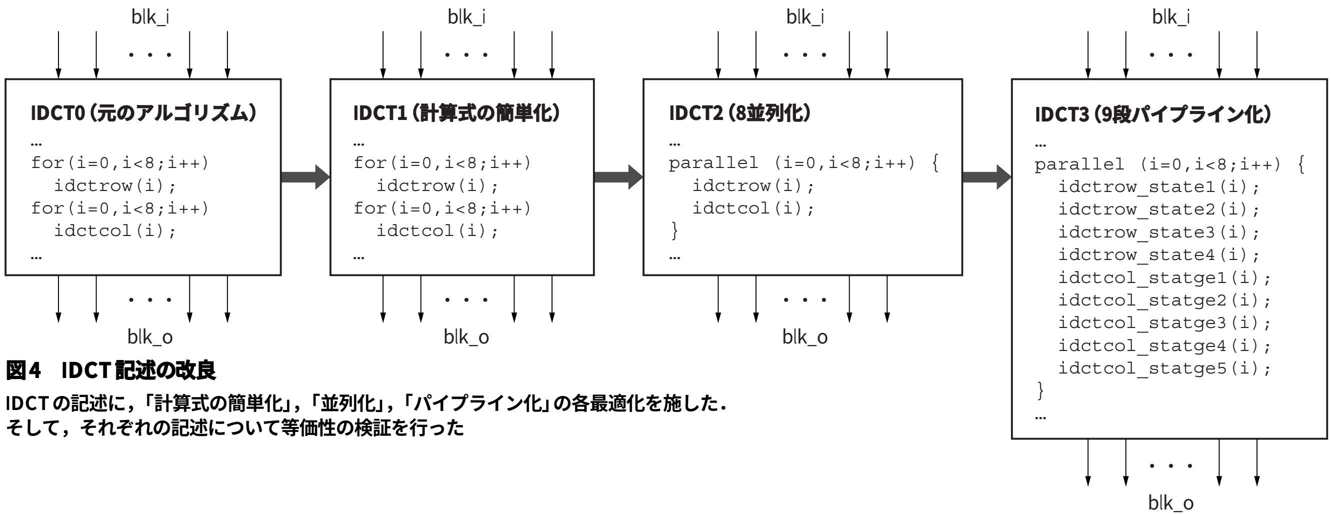


図4　IDCT記述の改良

IDCTの記述に，「計算式の簡単化」，「並列化」，「パイプライン化」の各最適化を施した．そして，それぞれの記述について等価性の検証を行った

る場合に同じ関数を利用することにすれば，SystemC言語で容易に表現できます．同じ種類の算術演算でも演算器が異なる場合には異なる関数を用意して利用します．

このような表現法を利用すると，IDCT1はRTLの設計記述になっているとも言え，先ほどの20～50秒程度の検証時間は，ビヘイビア記述とRTL記述の間の等価性検証にかかった時間であると言えます．ただし，この場合には，レイテンシやスループットをスケジューリングに合わせて指定してやる必要があります．

● ワード・レベルかビット・レベルかで処理時間が変わる

さて，この20～50秒という検証時間は，実はかなり高速です．次回で述べますが，IDCTは回路規模としては数万ゲートに相当します．例えばIDCTに対するRTL記述どうしの等価性検証には，同じCalypto社のツールで1～2時間程度かかります．これに比べると，今回の結果はかなり高速です．それはなぜでしょう？

それは，本ツールが等価性の判定を行うときに利用している手法が異なるからです．**リスト1**のIDCT0や**図4**に示すIDCT1，IDCT2，IDCT3などの間の等価性検証では，ワード・レベルの検証手法を使って等価性を判定できます．これに対してRTL記述どうし（片方がもう一方をリタイミングしたものになっている場合など）の等価性検証では，ワード・レベルの検証手法を適用しても完全に検証しきれません．ビット・レベルの検証手法を併用する必要があります（あるいは，ほとんどビット・レベルの検証手法を使わなければならない場合も多い）．

ワード・レベル検証手法では，**リスト1**の記述に出てくる整数変数が，それぞれ一つの変数として扱われます．これに対してビット・レベル検証手法では，整数変数は個々のビットに展開して処理されます．**リスト1**の場合はshortなので，一つの整数変数は16個のビット変数に変換されます．つまり，変数の数が16倍になります．

一般に，変数の数が増大すると，検証問題として急激に解くのが難しくなり（処理量が多くなり），等価性検証に要する時間に大きな差がでます．**リスト1**や**図4**の記述の差は，例えば**図5**(**a**)に示すようなものです．算術式の性質を利用すれば，a×(b＋c)とa×b＋a×cが等価であることは，a，b，cのビット幅によらず示すことができます．これをワード・レベル検証と呼んでいます．a，b，cが16ビット幅である場合に，a，b，cをa0，b0，c0からa15，b15，c15までに展開して，48個のビット変数の間の論理式の等価性という形で検証することも可能です．これをビット・レベル検証と呼びます．ビット・レベル検証の問題を解く処理量は，ワード・レベル検証の場合よりもはるかに多くなります．

図5(**b**)を見てください．これはリタイミングの例で，遅延時間が長い乗算器を二つに分け，2番目を次のクロックで計算するようにフリップフロップの右側に移し，結果として，各組み合わせ回路ブロックの信号遅延を均一化しようとしています．順序回路の等価性検証では，各クロックごとの変数やフリップフロップの値を解析することになります．この段階ではワード・レベルの情報は消えてしまっているため，ワード・レベル検証を適用することは不可能です．かならず各ビットに展開して，論理式としての等価性を検証する必要があります．

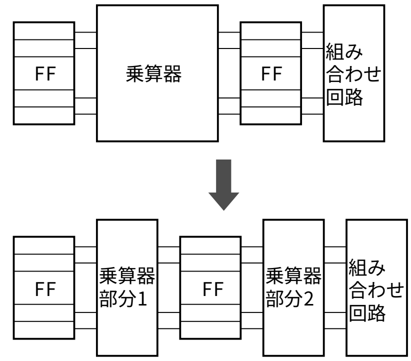
IDCTの例で，数十秒で検証が終わるのはワード・レベ

図5
ワード・レベル検証とビット・レベル検証
ワード・レベル検証では，算術式の性質を利用して，ワード・レベルのまま等価であることを示す．ビット・レベル検証では，変数をビット単位に展開し，ビット変数の間の論理式の等価性という形で検証する

a×(b+c) = a×b+a×c

- **ワード・レベル検証**
  分配法則を用いて，ワードごとに検証
- **ビット・レベル検証**
  a0，b0，c0，a1，b1，c1，…，a15，b15，c15の48個の変数からなる論理式に展開して検証

(**a**) 2種類の検証手法



(**b**) リタイミングの例

ル検証に相当する場合で，1，2時間かかるのはビット・レベル検証に相当する場合です．したがって，等価性検証を行う場合には，比較する設計記述間の相違から，ワード・レベルで検証できるのかビット・レベル検証が必要なのかが見極められると，検証時間をある程度予測できます．

ここで注意していただきたいことは，「設計全体をすべてワード・レベルで検証する」，「すべてビット・レベルで検証する」という二者択一ではないということです．設計の各部分（各記述ブロック）ごとに，ワード・レベルで検証できる場合やビット・レベル検証を併用しなければならない場合に分かれるということです．一般に，二つの設計記述が大きく異なると，ビット・レベルで検証しなければならなくなる可能性が高くなります．

したがって，**図4**の一連の最適化を設計に施す場合，個々の最適化ごとに等価性を検証することが，検証の効率化という面で非常に有効と言えます．

● 並列・パイプライン化してもワード・レベルで検証可能

さて，今度は**図4**のIDCT2について検証してみましょう．IDCT2はIDCT0やIDCT1の記述に現れる`idctrow`と`idctcol`をペアにし，引き数値が異なる八つの場合をすべて並列に実行するように変換した記述です．したがって，IDCT2は八つの並列プロセスからなるSystemC記述になっています．IDCT0とIDCT2の間の等価性検証は，本ツールを使って22秒くらいで行えます．単に並列化しただけなので，完全なワード・レベル検証で処理でき，高速に等価性を判定できます．

また，IDCT3は，IDCT2の`idctrow`を4段パイプラインに，`idctcol`を5段パイプラインに（合計9段に）した設計記述です．これも，基本的にパイプライン化だけの変換なので，完全なワード・レベルの検証が可能です．

IDCT0とIDCT3の等価性や，IDCT2とIDCT3の等価性は60秒程度で検証できます．いずれの場合も，上で述べたように前処理に15秒程度かかっているはずなので，検証時間を比較する場合にはそれを考慮してください．

以上のように，完全にワード・レベルで検証できる場合には，きわめて高速に等価性検証を行えます．今回のツールでは，検証実行時に（ワード・レベル検証ではなく）ビット・レベル検証をかならず行うように指定することも可能です．しかしこのようにすると，上記のIDCT0，IDCT1，IDCT2，IDCT3の等価性検証はかなり難しい問題となり，検証時間が大幅に増大します．

今回は，システム・レベル設計記述どうしの等価性検証について説明しました．ここで示した各種の変換は，ワード・レベル検証によって等価性を判定できる場合の例です．実際の設計では，プログラム的な設計記述をビヘイビア合成ツールが処理できる記述に人手で書き換える際にこうした変換がよく出てきます．「ビヘイビア合成ツールによって処理できない」，あるいは「処理できても効率的なRTL記述を生成できない」といった場合がよくあります．そのようなときに，しばしば人手で設計記述の書き換えを行います．その変換の正しさを保証する手段として，今回のような等価性検証は非常に有効であるといえます．

ふじた・まさひろ
東京大学 大規模集積システム設計教育研究センター（VDEC）教授

<筆者プロフィール>
藤田昌宏．1985年，東京大学大学院 情報工学専門課程修了．工学博士．同年，富士通研究所入社．論理設計用CAD技術の研究開発に従事．1993年より米国富士通研究所へ出向，CAD研究グループを立ち上げる．2000年3月より東京大学大学院 工学系研究科電子工学専攻 教授．2004年4月より東京大学 大規模集積システム設計教育研究センター（VDEC）教授．ディジタル・システム設計支援技術，とくにシステム・レベル合成，論理合成，論理検証を中心に研究している．